Making Discrete and the Interpenetration of Code and Language

Let us now shift from interpreting code through the worldviews of speech and writing to the inverse approach of interpreting speech and writing through the worldview of code. An operation scarcely mentioned by Saussure and Derrida but central to code is digitization, which I interpret here as the act of making something discrete rather than continuous, that is, digital rather than analog. The act of making discrete extends through multiple levels of scale, from the physical process of forming bit patterns up through a complex hierarchy in which programs are written to compile other programs. Understanding the practices through which this hierarchy is constructed, as well as the empowerments and limitations the hierarchy entails, is an important step in theorizing code in relation to speech and writing.

Let me make a claim that, in the interest of space, I will assert rather than substantiate: the world as we sense it on a human scale is basically analog. Over millennia, humans have developed biological modifications and technological prostheses to impose digitization on these analog processes, from the physiological evolution needed to produce speech to sophisticated digital computers. From a continuous stream of breath, speech introduces the discreteness of phonemes; writing carries digitization farther by adding artifacts to this physiological process, developing inscription technologies that represent phonemes with alphabetic letters. At every point, analog processes interpenetrate and cooperate with these digitizations. Experienced readers, for example, perceive words not as individual letters but as patterns perceived in a single glance. The synergy between the analog and digital capitalizes on the strengths distinctive to each. As we have seen, digitization allows fine-tuned error control and depth of coding, whereas analog processes tie in with highly evolved human capabilities of pattern processing. In addition, the analog function of morphological resemblance, that is, similarity of form, is the principal and indeed (so far as I know) the only way to convey information from one instantiated entity to a differently instantiated entity.

How do practices of making discrete work in the digital computer? We have already heard about the formation of the bit stream from changing voltages channeled through logic gates, a process that utilizes morphological resemblance. From the bit pattern bytes are formed, usually with each byte composed of eight bits—seven bits to represent the ASCII code, and an empty one that can be assigned special significance. At each of these stages, the technology can embody features that were once useful but have since become obsolete. For example, the ASCII code contains a seven-bit pattern corresponding to a bell ringing on a teletype. Although teletypes are no longer in use, the bit pattern remains because retrofitting the ASCII code to

delete it would require far more labor than would be justified by the benefit. To some extent, then, the technology functions like a rock strata, with the lower layers bearing the fossilized marks of technologies now extinct.

In the progression from speech to writing to code, each successor regime introduces features not present in its predecessors. In *Of Grammatology*, Derrida repeatedly refers to the space between words in alphabetic writing to demonstrate his point that writing cannot be adequately understood simply as the transcription of speech patterns (39, passim). Writing, he argues, exceeds speech and thus cannot be encapsulated within this predecessor regime; "writing is at the same time more exterior to speech, not being its 'image' or its 'symbol,' and more interior to speech, which is already in itself a writing" (46). Not coincidentally, spaces play an important role in the digitization of writing by making the separation of one word from another visually clear, thus contributing to the evolution of the codex book as it increasingly realized its potential as a medium distinct from speech. Similarly, code has characteristics that occur neither in speech nor in writing—processes that, by exceeding these legacy systems, mark a disjunction.

To explore these characteristics, let us now jump to a high level in the hierarchy of code and consider object-oriented programming languages, such as the ubiquitous C++. (I leave out of this discussion the newer languages of Java and C#, for which similar arguments could be made). C++ commands are written in ASCII and then converted into machine language, so this high-level programming language, like everything that happens in the computer, builds on a binary base. Nevertheless, C++ instantiates a profound shift of perspective from machine language and also from the procedural languages like FORTRAN and BASIC that preceded it. Whereas procedural languages conceptualize the program as a flow of modularized procedures (often diagrammed with a flowchart) that function as commands to the machine, object-oriented languages are modeled after natural languages and create a syntax using the equivalent of nouns (that is, objects) and verbs (processes in the system design).

A significant advantage to this mode of conceptualization, as Bruce Eckel explains in *Thinking in C++*, is that it allows programmers to conceptualize the solution in the same terms used to describe the problem. In procedural languages, by contrast, the problem would be stated in real-world terms (Eckel's example is "put the grommet in the bin"), whereas the solution would have to be expressed in terms of behaviors the machine could execute ("set the bit in the chip that means that the relay will close"; 43). C++ reduces the conceptual overhead by allowing both the solution and the problem to be expressed in equivalent terms, with the language's structure

performing the work of translating between machine behaviors and human perceptions.

The heart of this innovation is allowing the programmer to express her understanding of the problem by defining classes, or abstract data types, that have both characteristics (data elements) and behaviors (functionalities). From a class, a set of objects instantiate the general idea in specific variations—the nouns referred to above. For example, if a class is defined as "shape," then objects in that class might be triangle, circle, square, and so on (37–38). Moreover, an object contains not only data but also functions that operate on the data—that is, it contains constraints that define it as a unit, and it also has encapsulated within it behaviors appropriate to that unit. For example, each object in "shape" might inherit the capability to be moved, to be erased, to be made different sizes, and so on, but each object would give these class characteristics its own interpretation. This method allows maximum flexibility in the initial design and in the inevitable revisions, modifications, and maintenance that large systems demand. The "verbs" then become the processes through which objects can interact with each other and the system design.

New objects can be added to a class without requiring that previous objects be changed, and new classes and metaclasses can also be added. Moreover, new objects can be created through inheritance, using a preexisting object as a base and then adding additional behaviors or characteristics. Since the way the classes are defined in effect describes the problem, the need for documentation external to the program is reduced; to a much greater extent than with procedural languages, the program serves as its own description. Another significant advantage of C++ is its ability to "hide" data and functions within an object, allowing the object to be treated as a unit without concern for these elements. "Abstraction is selective ignorance," Andrew Koenig and Barbara E. Moo write in *Accelerated C++*, a potent aphorism that speaks to the importance in large systems of hiding details until they need to be known.[28] Abstraction (defining classes), encapsulation (hiding details within objects and, on a metalevel, within classes), and inheritance (deriving new objects by building on preexisting objects) are the strategies that give object-oriented programs their superior flexibility and ease of design.

We can now see that object-oriented programs achieve their usefulness principally through the ways they anatomize the problems they are created to solve—that is, the ways in which they cut up the world. Obviously a great deal of skill and intuition goes into the selection of the appropriate classes and objects; the trick is to state the problem so it achieves abstraction in an

appropriate way. This often requires multiple revisions to get it right, so ease of revision is crucial.

Some of the strategies C++ uses to achieve its language-like flexibility illustrate how it makes use of properties that do not appear in speech or writing and are specific to coding systems. Procedural languages work by what is called "early binding," a process in which the compiler (the part of the code hierarchy that translates higher-level commands into the machine language) works with the linker to direct a function call (a message calling for a particular function to be run) to the absolute address of the code to be executed. At the time of compiling, early binding thus activates a direct link between the program, compiler, and address, joining these elements before the program is actually run. C++, by contrast, uses "late binding," in which the compiler ensures that the function exists and checks its form for accuracy, but the actual address of the code is not used until the program is run.[29] Late binding is part of what allows the objects to be self-contained with minimum interference with other objects.

The point of this rather technical discussion is simple: there is no parallel to compiling in speech or writing, much less a distinction between compiling and run-time. The closest analogy, perhaps, is the translation of speech sounds or graphic letter forms into synapses in the human brain, but even to suggest this analogy risks confusing the *production* of speech and writing with its *interpretation* by a human user. Like speech and writing, computer behaviors can be interpreted by human users at multiple levels and in diverse ways, but this activity comes after (or before) the computer activity of compiling code and running programs.

Compiling (and interpreting, for which similar arguments can be made) is part of the complex web of processes, events, and interfaces that mediate between humans and machines, and its structure bespeaks the needs of both parties involved in the transaction. The importance of compiling (and interpreting) to digital technologies underscores the fact that new emphases emerge with code that, although not unknown in speech and writing, operate in ways specific to networked and programmable media. At the heart of this difference is the need to mediate between the natural languages native to human intelligence and the binary code native to intelligent machines. As a consequence, code implies a partnership between humans and intelligent machines in which the linguistic practices of each influence and interpenetrate the other.[30]

The evolution of C++ grew from precisely this kind of interpenetration. C++ is consciously modeled after natural language; once it came into wide use, it also affected how natural language is understood. We can see this

two-way flow at work in the following observation by Bruce Eckel, in which he constructs the computer as an extension of the human mind. He writes, "The genesis of the computer revolution was in a machine. The genesis of our programming languages thus tends to look like that machine. But the computer is not so much a machine as it is a mind amplification tool and a different kind of expressive medium. As a result, the tools are beginning to look less like machines and more like parts of our minds, and more like other expressive mediums like writing, painting, sculpture, animation or filmmaking. Object-oriented programming is part of this movement toward the computer as an expressive medium" (*Thinking in C++*, 35). As computers are increasingly understood (and modeled after) "expressive mediums" like writing, they begin to acquire the familiar and potent capability of writing not merely to express thought but actively to constitute it. As high-level computer languages move closer to natural languages, the processes of intermediation by which each affects the other accelerate and intensify. Rita Raley has written on the relation between the spread of Global English and the interpenetration of programming languages with English syntax, grammar, and lexicon.[31] In addition, the creative writing practices of "codework," practiced by such artists as MEZ, Talan Memmott, Alan Sondheim, and others, mingle code and English in a pastiche that, by analogy with two natural languages that similarly intermingle, might be called a creole.[32]

The vectors associated with these processes do not all point in the same direction. As explored in chapter 8, (mis)recognizing visualizations of computational simulations as creatures like us both anthropomorphizes the simulations and "computationalizes" the humans. Knowing that binary code underlies complex emergent processes reinforces the view that human consciousness emerges from similar machinic processes, as explored in chapter 7. Anxieties can arise when the operations of the computer are mystified to the extent that users lose sight of (or never know) how the software actually works, thus putting themselves at the mercy of predatory companies like Microsoft, which makes it easy (or inevitable) for users to accept at face value the metaphors the corporation spoon-feeds them, a concern explored in chapter 6. These dynamics make unmistakably clear that computers are no longer merely tools (if they ever were) but are complex systems that increasingly produce the conditions, ideologies, assumptions, and practices that help to constitute what we call reality.

The operations of "making discrete" highlighted by digital computers clearly have ideological implications. Indeed, Wendy Hui Kyong Chun goes so far as to say that *software is ideology*, instancing Althusser's definition of ideology as "the representation of the subject's imaginary relationship to his

or her real conditions of existence."[33] As she points out, desktop metaphors such as folders, trash cans, and so on create an imaginary relationship of the user to the actual command core of the machine, that is, to the "real conditions of existence" that in fact determine the parameters within which the user's actions can be understood as legible. As is true for other forms of ideology, the interpolation of the user into the machinic system does not require his or her conscious recognition of how he or she is being disciplined by the machine to become a certain kind of subject. As we know, interpolation is most effective when it is largely unconscious.

This conclusion makes abundantly clear why we cannot afford to ignore code or allow it to remain the exclusive concern of computer programmers and engineers. Strategies can emerge from a deep understanding of code that can be used to resist and subvert hegemonic control by megacorporations;[34] ideological critiques can explore the implications of code for cultural processes, a project already evident in Matthew Fuller's call, seconded by Matthew Kirschenbaum, for critical software studies;[35] readings of seminal literary texts can explore the implications of code for human thought and agency, among other concerns. Code is not the enemy, any more than it is the savior. Rather code is increasingly positioned as language's pervasive partner. Implicit in the juxtaposition is the intermediation of human thought and machine intelligence, with all the dangers, possibilities, liberations, and complexities this implies.